# Analysis of Cache Invalidation Patterns in Multiprocessors

**Wolf-Dietrich Weber and Anoop Gupta**
(Draft: Sep 20, 1988)
Computer Systems Laboratory
Stanford University, CA 94305

**Abstract**

To make shared-memory multiprocessors scalable, researchers are now exploring cache coherence protocols that do not rely on broadcast, but instead send invalidation messages to individual caches that contain stale data. The feasibility of such directory-based protocols is highly sensitive to the cache invalidation patterns that parallel programs exhibit. In this paper, we analyze the cache invalidation patterns caused by several parallel applications and investigate the effect of these patterns on a directory-based protocol. Our results are based on multiprocessor traces with 4, 8 and 16 processors. To get insight into what the invalidation patterns would look like beyond 16 processors, we propose a classification scheme for data objects found in parallel applications and link the invalidation traffic patterns observed in the traces back to these high-level objects. Our results show that synchronization objects have very different invalidation patterns from those of other data objects. A write reference to a synchronization object usually causes invalidations in many more caches. We point out situations where restructuring the application seems appropriate to reduce the invalidation traffic, and others where hardware support is more appropriate. Our results also show that it should be possible to scale "well-written" parallel programs to a large number of processors without an explosion in invalidation traffic.

## 1 Introduction

One of the most critical issues in the design of shared-memory multiprocessors is the cache coherence strategy. Most multiprocessors rely on a shared bus and use a broadcast-based protocol to keep the caches coherent [8,16,18,15,23]. However, such multiprocessors are not very scalable, as the shared-bus soon becomes a bottleneck. As an alternative, researchers have started exploring cache coherence protocols that do not rely on broadcast, the most common example being directory-based protocols [2,4]. These protocols rely on the system having knowledge about which caches contain a particular piece of data. On a write, invalidation messages are sent only to these specific caches. The number of pointers in each directory entry determines how many other caches can be kept track of. Determining the performance of directory-based protocols requires the answer to several questions. We would like to know the distribution of the number of remote caches that need to be invalidated on shared writes. We would like to know how these distributions scale as the number of processors is increased. We are interested in knowing what types of data objects in the applications result in what kind of invalidation patterns. This paper attempts to answer some of these questions for directory-based protocols.

We analyze the patterns of invalidation traffic produced by a set of five application programs. Three of the five applications selected are "real" parallel programs, in the sense that they solve real-world problems and that a lot of effort has gone into obtaining good processor efficiency with them. The remaining two applications are smaller, but they are still interesting in that they could form the kernels of larger applications. Our study is based on memory reference traces obtained for the applications when simulating 4, 8, and 16 processors.[1] The traces were

[1] Previous studies [1,2] presented results using traces with only 4 processors. This study uses a more extensive set of applications, a larger number of processors, and goes more deeply into the causes of invalidations patterns.



To appear in Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, April 1989.

generated using software-traps on a 4-processor VAX-8350 and a VAX-3200 running MACH. In addition to presenting the invalidation patterns as observed directly from the traces, the paper links the invalidation patterns to the high-level program data structures (objects) that cause them. A classification of such shared objects on the basis of their expected invalidation behavior is given. Linking the invalidation patterns to the high-level objects helps us predict how the invalidation traffic would change as the number of processors is increased. It is far more accurate to extrapolate the behavior of each class of data object than to simply extrapolate the composite behavior. For the application types we have considered, our results indicate that it is quite possible to write parallel programs that do not create an enormous amount of invalidation traffic. Thus directory-based schemes with just a few pointers per entry could efficiently execute well-designed parallel programs.

The next section explains the methodology used in generating the traces and explains how the traces were analyzed. Section 3 introduces the five applications used in this study and gives a brief overview of their computational behavior. In Section 4 we present some basic trace characteristics. In the next section we present the proposed classification of shared data objects in parallel programs. Section 6 goes into a detailed analysis of the invalidation behavior of each application and relates these patterns to specific data objects in the applications. Section 7 assembles the results from the various applications and presents conclusions.

## 2 Methodology and Assumptions

The traces were collected using a combined hardware/software method [7]. The process creation is modified to have one master process, which controls the actual tracing, and a number of slave processes, one for each "virtual processor". Once the desired start position for tracing is reached, each of the slaves stops itself and is then single-stepped by the master. The stepping takes place in a round-robin fashion. The stepping employs the UNIX *ptrace* system call which uses the T-bit on the VAX. While stepping, the master process records data in the trace file. For each reference, the type (I-fetch, read, or write), the address, and the CPU number are recorded. Trace lengths used were 20Mbytes for 4-processor traces, 30Mbytes for 8-processor traces, and 50Mbytes for 16-processor traces. This corresponds to about 2.5, 4 and 7 million references respectively, or around 0.5 million references per processor.

The traces were gathered on a VAX-8350 with 4 processors and a VAX-3200 workstation, both running the MACH operating system. MACH allows allocation of shared memory for the processors. On the 8350 it takes about 24 hours to obtain 20Mbytes of trace, while the VAX-3200 can gather about 50Mbytes in the same time.

Once the traces were gathered, they were used as input to a program that simulates multiprocessor cache behavior and gathers statistics. Infinite caches were used for simplicity of the cache simulator. The cache coherence protocol used was an invalidation scheme similar to the Berkeley Ownership scheme [16]. For each potential invalidation, a record was written containing the CPU number, the data address, the most recent instruction address and the number of other caches actually invalidated. The data and instruction addresses were later used to associate the invalidation with the high-level language construct that caused it. Several post-processing programs were used to gather statistics from the invalidation traces.

The main advantage of the software scheme of gathering traces is that we can get traces for an arbitrary number of processors, which is not possible with hardware schemes like ATUM [20]. However, there are some disadvantages too. For example, the *ptrace* call does not trace operating system calls, but rather treats them as a single reference. This is not a major problem

in this study, since there are not many operating system calls in the sections traced. Also, each instruction takes one time unit to complete, regardless of the complexity of the instruction. This is clearly an oversimplification, but there is no reason to believe that it significantly distorts results.

# 3 Application Programs

In this section we describe the data structures and computational behavior of the applications. This is important background for Section 6, where we relate invalidation traffic to high-level objects. The applications used for tracing were selected to represent a variety of algorithms used in an engineering computing environment. All of the applications were written in C. The Argonne National Laboratory macro package [11,12] was used to provide synchronization and sharing primitives. The synchronization primitives used include spin locks, as well as barriers and distributed loops.

## 3.1 Maxflow

Maxflow [3] finds the maximum flow in a directed graph. This is a common problem in operations research and many other fields. The program is a parallel implementation of an algorithm proposed by Goldberg and Tarjan. The bulk of execution time is spent picking off nodes from a task queue, adjusting the flow along its incoming and outgoing edges, and then placing its successor nodes onto a task queue. Maxflow exploits parallelism at a fine grain.

Maxflow does not assign the nodes of the graph to processors statically. Instead, task queues are used to distribute the load. Each processor has its own local task queue and need only go to the single global task queue when the local queue is empty. Tasks are put onto the global queue only when processes are waiting there, and onto the local queue otherwise. Note that the task queues are made up of the nodes themselves, linked together with appropriate pointers. Locks are used to serialize access to each node element, but contention for these is fairly low, as there are many more nodes than processors. In Section 6 we will see that most invalidations are related to the global task queue and the migration of node data from one processor to another.

The traces were collected while solving Maxflow for a set of nodes arranged as a 10-ary 2-cube. Tracing was started as the program entered the main loop after completing the initial distance labeling. The implementation provides speedups of about 8 with 12 processors.

## 3.2 SA-TSP

SA-TSP [21] solves the traveling salesperson problem using simulated annealing [10]. A linear array contains the cities in tour order. At each step, a processor selects a pair of cities to swap. The swap is performed if it results in a shorter tour or if the increase in tour distance is within the margin prescribed by the cooling function. The tour is locked *only* during the actual swap, which means that errors occur when the tour has been modified between making the decision and actually performing the swap. This trades off quality of solution for greater speedup. Note that there is only one global lock for all the tour data. This becomes a major bottleneck as the number of processors increases. In the initial annealing phase – which is the section we traced – most moves are accepted and contention for the lock is especially large. While the program achieves an overall speedup of 7 with 8 processors, no more than 4 processors can be kept busy during this initial portion.

### 3.3 MP3D

MP3D [13,14] is a 3-dimensional particle simulator for rarified flow. It is used to study the shock waves created as an object flies at high speed through the upper atmosphere. MP3D is a good example of scientific code that is vectorizable and can be parallelized using distributed loops. A version of MP3D that runs on the Cray-2 is being used extensively at NASA for research.

The overall computation of MP3D consists of evaluating the positions and velocities of molecules over a sequence of time steps. During each time step, the molecules are picked up one at a time and moved as governed by their velocity vectors. Collisions with the boundaries and with each other are resolved. The simulator is well suited to parallelization because each molecule can be treated independently at each time step. The work is spread over the processors with the help of a distributed loop, consisting of a lock and a global index variable. Each processor obtains the lock, reads the index, increments it, and releases the lock. In this manner the processes pick up the index of the next particle to be moved. The traces cover most of one time step, i.e. each particle is moved once. No locking is employed in the various arrays that keep track of the particles and space, because collisions are impossible in the particle arrays and very rare in the space arrays. Thus, the distributed loop is the only synchronization seen in this trace.

### 3.4 Distributed CSIM

Distributed CSIM [22] is a parallel logic simulator developed at Stanford University. It is an interesting application based on the Chandy-Misra simulation algorithm [5], which is specially designed for highly parallel machines — unlike event-based algorithms, this algorithm does not rely on a single global time during simulation.

The primary data structures associated with the simulator are the logic elements (e.g., AND-gates, flip-flops), the nets (the wires linking the elements) and the task queues which contain activated elements. Each processor has as many task queues as there are other processors. This ensures that there is no contention when adding elements to some other processor's queue. Each processor executes the following loop. It removes an activated element from one of its task queues and determines the changes on the element's outputs. It then looks up the net data structure to determine which elements are affected by the output change and potentially schedules those activated elements onto other processors' task queues. Newly activated elements are assigned to other processors in a round-robin fashion.

### 3.5 LocusRoute

LocusRoute [17] is a global router for VLSI standard cells. It is a real application in that it is a part of a system that has been used to design real integrated circuits, and it has been highly tuned to run well on a shared-memory multiprocessor. LocusRoute represents the class of parallel programs that exploit fairly coarse grain parallelism.

The LocusRoute program exploits parallelism by routing multiple wires in a circuit concurrently. Each processor executes the following loop: (i) remove a wire to route from the task queue; (ii) explore alternative routes; and (iii) pick the best route for the wire and place it there. The central data structure used in LocusRoute is a grid of cells called the *cost array*. Each row of the cost array corresponds to a routing channel for standard cells. LocusRoute uses the cost array to record the presence of a wire at each point, and the congestion of a route is used

as a cost function for guiding the placement of new wires. No locking is needed in the cost array, which is accessed and updated simultaneously by several processors, because the effect of occasional collisions is tolerable. Each routing task is fairly large grain, which prevents the task queue from becoming a bottleneck.

# 4 Trace Characteristics

Table 1 gives an overview of the traces of the five applications. For each application, we give the trace length in number of references and the breakdown in terms of I-fetches, reads and writes. We also show the proportion of shared writes, and the average number of invalidations caused by each shared write. In addition to absolute numbers, the columns also list the number of references in each category as a fraction of all references in the trace.

Table 1: General Trace Characteristics.

| Application | num of CPUs | refs mill | I-fetches mill | I-fetches % | reads mill | reads % | writes mill | writes % | shared writes thous | shared writes % | avg. invals per sh-wrt |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Maxflow | 4 | 2.62 | 1.21 | 46 | 1.06 | 40 | 0.35 | 13 | 73.6 | 2.81 | 0.30 |
| | 8 | 4.15 | 1.91 | 46 | 1.69 | 41 | 0.55 | 13 | 121.6 | 2.93 | 0.49 |
| | 16 | 8.36 | 3.86 | 46 | 3.46 | 41 | 1.04 | 12 | 274.8 | 3.29 | 1.07 |
| SA-TSP | 4 | 2.65 | 1.10 | 42 | 1.12 | 42 | 0.43 | 16 | 19.5 | 0.74 | 1.27 |
| | 8 | 4.16 | 1.84 | 44 | 1.88 | 45 | 0.44 | 11 | 37.3 | 0.90 | 2.29 |
| | 16 | 7.11 | 3.30 | 46 | 3.37 | 47 | 0.43 | 6 | 77.0 | 1.08 | 2.93 |
| MP3D | 4 | 2.53 | 1.57 | 62 | 0.80 | 32 | 0.17 | 7 | 83.7 | 3.31 | 0.68 |
| | 8 | 3.59 | 2.22 | 62 | 1.13 | 31 | 0.23 | 6 | 119.9 | 3.34 | 0.80 |
| | 16 | 7.05 | 4.28 | 61 | 2.33 | 33 | 0.43 | 6 | 230.3 | 3.27 | 1.03 |
| Dist CSIM | 4 | 2.61 | 1.28 | 49 | 1.01 | 39 | 0.32 | 12 | 8.5 | 0.33 | 0.44 |
| | 8 | 4.13 | 2.04 | 49 | 1.61 | 39 | 0.48 | 12 | 19.9 | 0.48 | 0.46 |
| | 16 | 7.09 | 3.52 | 50 | 2.80 | 39 | 0.77 | 11 | 42.5 | 0.60 | 0.51 |
| LocusRoute | 4 | 2.60 | 1.31 | 50 | 0.95 | 37 | 0.33 | 13 | 5.6 | 0.22 | 0.56 |
| | 8 | 4.34 | 2.26 | 52 | 1.59 | 37 | 0.49 | 11 | 4.8 | 0.11 | 1.07 |
| | 16 | 7.70 | 3.95 | 51 | 2.83 | 37 | 0.92 | 12 | 9.2 | 0.12 | 1.28 |

In all of the programs, with the exception of MP3D, about 45-50% of the references are I-fetches. MP3D has a larger proportion of I-fetches because there are a lot of array references which require several instructions to compute the effective address of the reference.

The proportion of read references varies from about 30% in MP3D to over 45% in SA-TSP. In SA-TSP there are a lot of simple integer reads when determining the effect of a swap on tour distance. The read fraction is low in MP3D because of the larger proportion of I-fetches.

Writes hover around 10-15% of all references. MP3D again stands out with a very low write fraction, again due to frequent array references. The number of writes in SA-TSP stays virtually constant even though the number of references increases greatly as we move from 4 to 16 processors. This is explained by the fact that writes are only used when a swap is accepted. Contention for the lock in the portion of SA-TSP traced is so large that no more swaps are accepted in the 16-processor trace than in the 4-processor trace. This portion of SA-TSP was

chosen to demonstrate the effects that a poorly written program segment may have on directory-based coherence schemes. Details are presented in Section 6.2.

In our study. we define *shared locations* to be those that are referenced by more than one process in the trace, and we define *shared writes* to be write references to shared locations. Note that some locations that really are shared in the application are considered not-shared in our study. because within the limited length of the trace multiple processes do not reference those locations.

The second to last column in Table 1 presents the proportion of shared writes in the applications — it is important to study shared writes because they can cause invalidations in some or all of the caches. There is a general trend towards an increasing percentage of shared writes as the number of processors increases. One reason for this is larger contention over locks. The locks are implemented as test-test&set sequences and thus cause additional shared writes when several processors are contending for a lock that was just freed. Also. as more processors are added. the chances of a data item being accessed by more than one process increases.[2] resulting in a larger fraction of shared writes.

An important metric of invalidation traffic is the average number of invalidations per shared write. The values are shown in the last column of Table 1. This parameter is the largest for SA-TSP, mostly due to invalidation traffic caused by the single global spin-lock. In fact. the average number of invalidations increases steeply with more processors due to the increased contention for this global lock. The number of invalidations per shared write is the smallest for distributed CSIM, and hardly goes up as the number of processors is increased. This is mainly because there are no synchronization objects in the portion of distributed CSIM traced. Averages. however. do not carry all of the interesting information. Consequently, the detailed invalidation distributions and their analysis are presented in Section 6.

# 5 Classification of Data Objects

When trying to extrapolate invalidation behavior to a larger number of processors. it is important to explain the invalidation patterns in terms of the underlying high-level structures which cause the invalidations. We distinguish several types of shared objects on the basis of their significance in parallel programs and their expected invalidation behavior [1]:

1. Code and read-only data objects.
2. Migratory objects.
3. Synchronization objects.
4. Mostly-read objects.
5. Frequently read/written objects.

Code and read-only data objects obviously do not cause invalidations at all. and thus pose no problem to any coherence scheme. A fixed database such as the matrix that contains the distances between cities in SA-TSP is a good example of such read-only data.

[2]This is partly because we get a longer trace for a run with more processors. and partly because with a larger number of processors. there is a higher probability that subtasks sharing data get scheduled on different processors rather than on the same processor.

Migratory data objects are those that are manipulated by only a single processor at a time. Shared objects protected by locks often exhibit this property. While such an object is being manipulated by one processor, the object's data resides in the associated cache. When the object is later manipulated by some other processor, the cache entry of the previous processor needs to be invalidated.[3] Migratory data usually causes a high proportion of *single* invalidations. The nodes in Maxflow are a good example of migratory data. Each node is evaluated by several processors over the complete run, but there is only one processor manipulating each node at any one time.

Synchronization objects such as locks can cause a very large number of invalidations if used improperly. When locks are implemented as test-test&set, and there are processors waiting on a lock, invalidations are caused each time the lock changes hands. As a lock is freed, all waiting processors fall through the test part of the test-test&set. They then attempt the test&set, but only one of them succeeds, causing invalidations in all other waiting processors' caches. It is important to use locks in a manner that minimizes contention for them.

An example of mostly-read data is the cost-array of LocusRoute. Most of the time it is just read, but every now and then, when the best route for a wire is decided, the array is written. It is a candidate for large number of invalidations because many reads by different processors occur before each write. Thus the data is cached by many processors, and a write causes many invalidations. However, since only the writes cause invalidations and writes are infrequent, the overall number of invalidations will be quite small.

Finally, there is frequently read/written data.[4] An example is the variable that counts how many processors are waiting on the global task queue in Maxflow. Frequently read/written data has the worst invalidation behavior. Unlike mostly-read objects, this data is written quite frequently. Although each write may only cause 3 or 4 invalidations, this may exceed the number of pointers per entry in a directory scheme, thus causing frequent broadcasts. This type of data object should be avoided if at all possible.

# 6 Application Case Studies

In this section we present the results of the detailed analysis of the invalidation traces produced when running the cache simulator over the multiprocessor traces. For each application, we show the overall invalidation patterns, the high-level objects causing the invalidations, the expected broadcast behavior of directory-based cache coherency schemes [4,2], and the scalability of the application beyond 16 processors.

The overall invalidation behavior is presented in terms of an invalidation distribution graph as shown in Figure 1. The graph shows the fraction of shared writes that caused no invalidations, single invalidations and so on. Ideally these graphs will contain a large proportion of small invalidations, as these can be handled efficiently by directory-based cache schemes. By comparing the invalidation distributions for 4, 8 and 16 processor traces, we can begin to get a feeling for how the invalidations scale with a larger number of processors. We would prefer to see no change in the distribution as the number of processors is increased, but it is more likely that a shift towards both more and larger invalidations occurs.

For each application, we also present another kind of graph that shows the fraction of broadcasts required as a function of the number of pointers per entry in the directory (see Figure

[3] Cheriton discusses a programming model based on such objects, called "*workforms*" in [6].

[4] Frequently read/written should be interpreted as both frequently read *and* frequently written.

6). A directory-based scheme such as $Dir_iB$ needs to use broadcast when a shared write is to a location that is contained in more caches than there are directory pointers for that entry. The data is plotted for directories with pointers varying from 1 to n, where n is the number of processors in the trace. We do not show directory schemes with 0 pointers as these require a broadcast for every shared write. Obviously, a directory with n pointers can keep track of all processors and broadcast is never required.

## 6.1 Maxflow

Figures 1, 2 and 3 show the invalidation distributions for Maxflow with 4, 8 and 16 processors respectively. Note that the distribution shifts to larger number of invalidations as the number of processors is increased. While at 4 processors only about 2% of shared writes cause more than one invalidation, this figure moves up to 18% with 16 processors. Analysis shows that the bulk of this increase is due to synchronization traffic involving the global task queue. Figures 4 and 5 show the invalidation distributions broken down by global queue traffic and all other invalidation traffic respectively. The global queue traffic includes all writes to the queue locks as well as the count of the number of processors blocked and the queue head pointer. It is clear that most of the spreading of the invalidation distribution is due to global-queue-related traffic.

A large fraction of the invalidations in Figures 1, 2 and 3 are single invalidations. They are caused by the manipulation of nodes and edges, which are good examples of migratory data objects. One processor picks up an active node and pushes flow through it. Later the node will get re-activated, and some other processor will pick it up and start processing it.

Some parameters of the nodes, such as its distance label, behave like mostly-read objects. Distance labels only get changed in the infrequent re-labeling steps. Between re-labeling, many processors may read a node's distance label causing re-labeling to generate a large number of invalidations. In the 16-processor trace, an average of 4.6 invalidations occur for each re-labeling write. Although 4.6 invalidations per shared write is large, the effect of these writes on the total number of invalidations is small since the writes are very infrequent.

The locks for the global task queue cause a large number of invalidations. Not only are they accessed and written frequently, but they also cause an average of about 2 invalidations per shared write in the 16-processor trace. The global queue is the major source of double or larger invalidations and should be a primary target for efforts aimed at improving the program.

The per-node locks, on the other hand, work well. They are an example of a synchronization object that causes few invalidations. There are so many more nodes than processors that contention is very limited.

The count of how many processors are waiting for the global task queue is checked frequently by all processors. It is also written frequently, namely whenever a process starts waiting on the global task queue. It is thus often read and written and causes many invalidations. It has an average of 2.8 invalidations per shared write and the highest number of shared writes to any single data object except for the global task queue locks.

A pattern of double invalidations found in Maxflow is very common when dealing with queues and is seen in several other applications. In Maxflow, one processor puts a node onto the global task queue, a second one picks it off, and a third one may later place the node on another queue. At first, the object is owned by one processor. When the node is picked up by the second processor, it becomes read-shared. Finally, the third processor writes the object, causing double invalidations in the link pointers. Many variations of this basic theme exist. Another example was found in POPS [9], a parallel rule-based expert system, where a single buffer is used for a

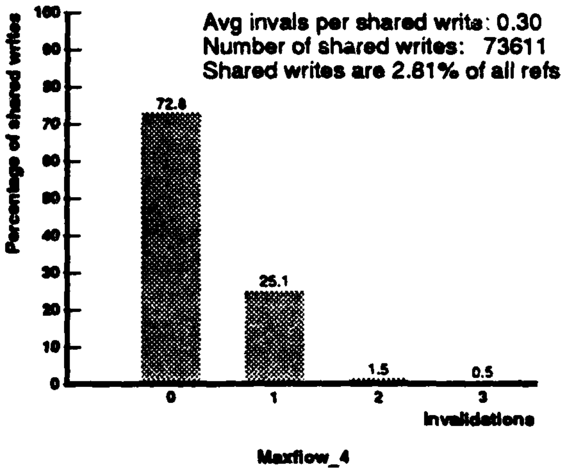


Figure 1: Maxflow 4

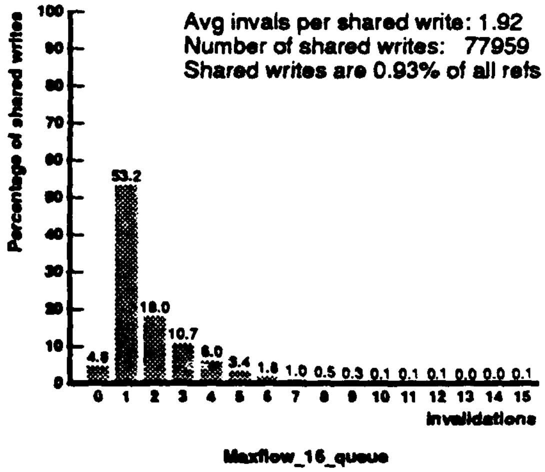


Figure 4: Maxflow 16 (Queue)

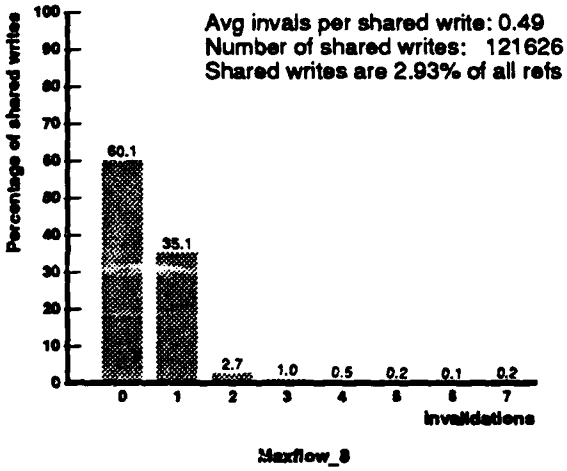


Figure 2: Maxflow 8

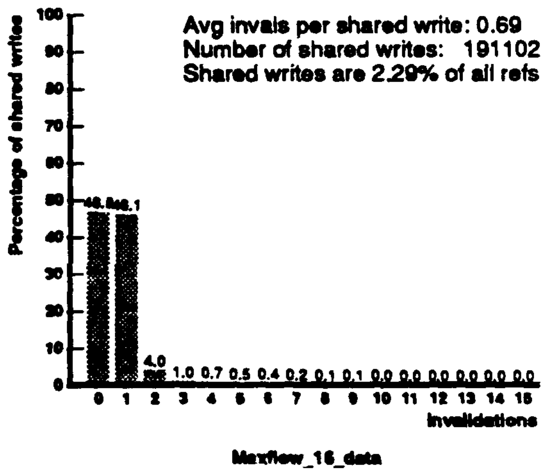


Figure 5: Maxflow 16 (Data)

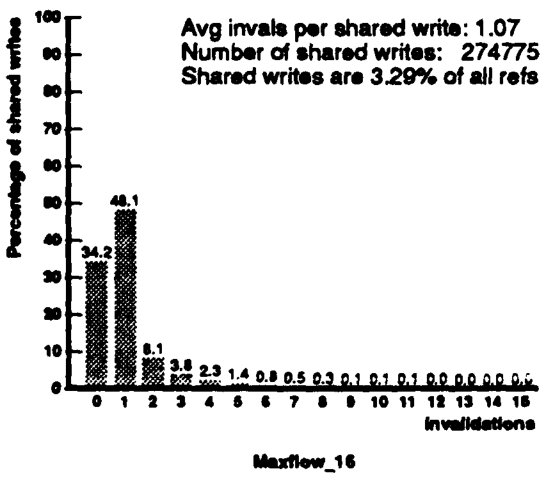


Figure 3: Maxflow 16

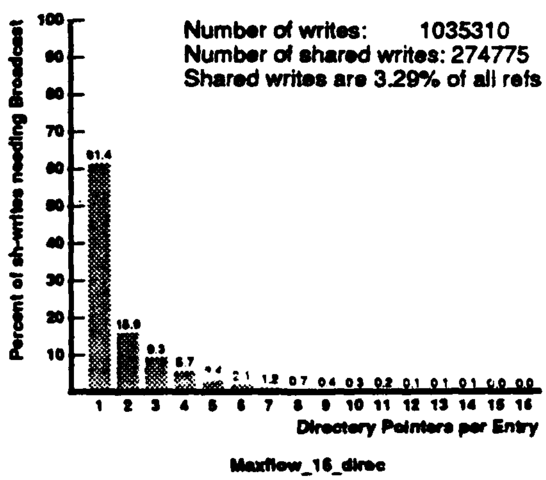


Figure 6: Maxflow 16 Directory Performance

task queue. An item is written into the buffer by one processor and read by another. Later. a third processor overwrites that item with some new data. thus invalidating the caches of both previous processors.

Figure 6 shows the proportion of shared writes that need to be broadcast for directory-based schemes with a varying number of pointers per entry. Although a scheme with two pointers per entry ($Dir_2B$ in [2]) only needs to broadcast 1.8% of shared writes with 4 processors. this figure jumps up to 15.9% for 16 processors. The invalidation distribution keeps spreading out as the number of processors is increased. mostly due to the invalidations associated with the global queue.

Let us now use the object classification to see how the invalidation distributions will change as the number of processors is scaled. We expect little change in the invalidations produced by migratory objects which will continue to produce single invalidations. Mostly-read objects will have a slightly higher average number of invalidations per shared write because more processors are likely to have cached the data. Note though. that the average number of invalidations per write (4.6 for 16 processors) may already be beyond the number of pointers stored in the directory. so no additional broadcasts will result. Synchronization objects and frequently read/written objects. on the other hand. are expected to have a higher average number of invalidations per shared write. In addition. we expect to see *more* shared writes due to synchronization. Since both synchronization objects with high contention and frequently read/written objects exist in Maxflow. we will see a continued spread of the invalidation distribution towards larger invalidations per shared write. If the program is to be scaled successfully. we will have to reduce synchronization contention and eliminate frequently read/written objects.

## 6.2 SA-TSP

Figures 7. 8 and 9 show the invalidation distributions for SA-TSP with 4, 8 and 16 processors. Most noticeable is the hump in the invalidation distribution for 16 processors at around 12 to 13 invalidations. This hump is less obvious with 8 processors and does not appear with 4 processors. All of the invalidations that make up this hump in the 16-processor distribution are due to the single global lock. In fact as many as 94% of all invalidations are due to that lock.

Figures 10 and 11 show the invalidation distribution for the 16-processor trace. broken down into lock traffic and all other data traffic. These graphs show clearly that nearly all of the large invalidations are due to the single lock. This is a good example of how a poorly-used lock can flood a machine with invalidations. In the initial annealing phase (the portion that was traced). most moves get accepted. Thus all of the processors want to update the global tour. which requires the lock. This results in very high contention for the lock. We found that with 12 to 13 processors waiting for the lock to be released. this phase of the program could use no more than about 4 processors. As the cooling function progresses. fewer and fewer moves are accepted. contention for the lock subsides and the program achieves good speedup.

The invalidations due to the shared data range between 0 and about 8. All of these are from the array that holds the order of the cities in the tour. The large average of shared-write invalidations is due to the mostly-read nature of this data. A processor needs to look at two cities and their four neighbors to determine whether a swap is to occur. and only if the swap meets certain annealing criteria does it actually take place. This means that for each proposed swap. at least four cities are only read. not written. Each successful swap thus invalidates a large number of caches. The frequency of invalidations is due to the fact that there are relatively few data objects (36 in this case. as the program was solving a tour with 36 cities). especially when

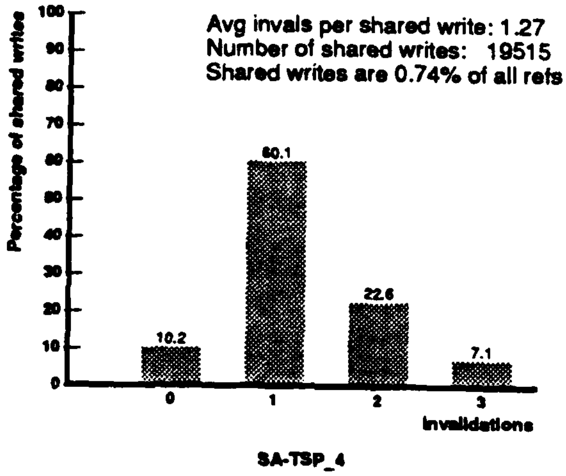


Figure 7: SA-TSP 4

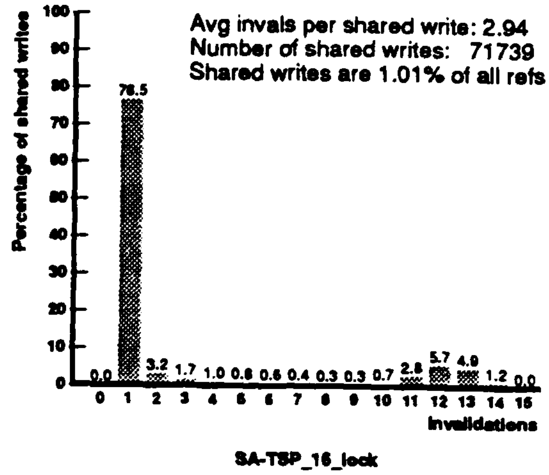


Figure 10: SA-TSP 16 (Lock)

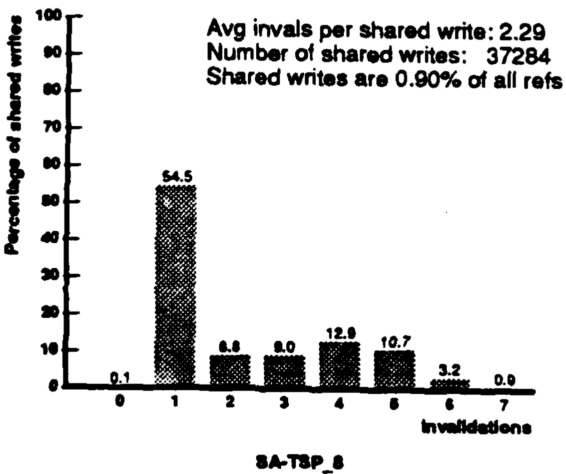


Figure 8: SA-TSP 8

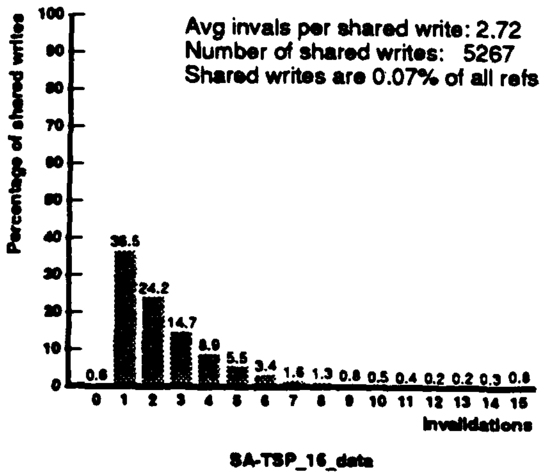


Figure 11: SA-TSP 16 (Data)

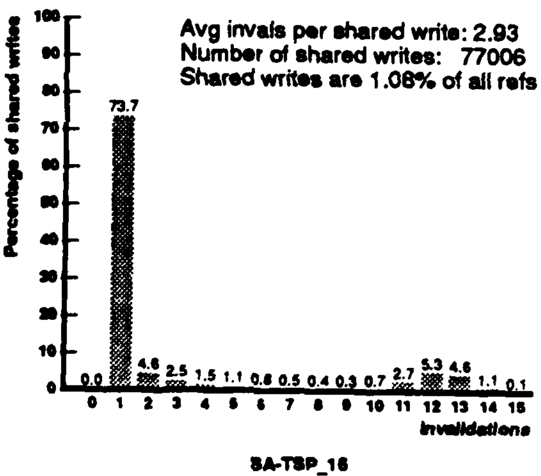


Figure 9: SA-TSP 16

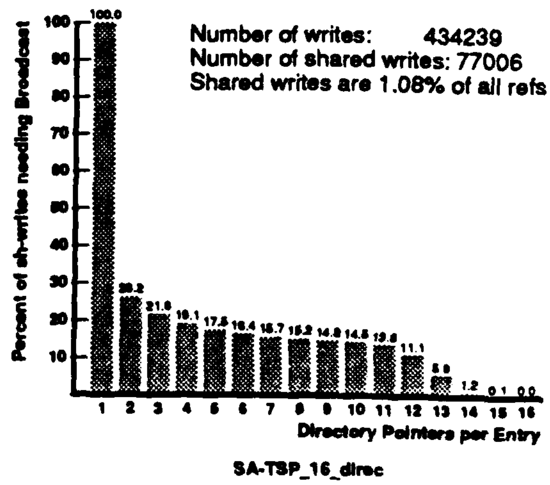


Figure 12: SA-TSP 16 Directory Performance

compared to LocusRoute or MP3D, where there are thousands of objects. Hence the chances of some other processor caching an object before it is written are much larger.

Figure 12 shows that even directory schemes with large number of pointers per entry perform poorly in the face of SA-TSP's invalidation traffic. After an initial lowering in the number of broadcasts with increasing number of directory pointers, the graph basically flattens out until we reach the hump. In the 16-processor case, a 10-pointer scheme would perform essentially as poorly as a 5-pointer scheme.

Further scaling of the number of processors would result in even larger contention for the global lock. This would move the invalidation hump to a larger number of invalidations per shared write. Essentially no additional useful work would be accomplished. A distributed locking scheme could reduce contention for the elements of the global tour. Even if the synchronization traffic is eliminated, however, we will still have a fair amount of shared data invalidation traffic. This is due to the fact that there are only a small number of data objects that are continuously read and written by several processors.

### 6.3 MP3D

Figures 13, 14 and 15 show the invalidation distributions for MP3D with 4, 8 and 16 processors respectively. The distributions are dominated by zero and single invalidations. As we increase the number of processors, some invalidations of 2 or more start to appear. This effect is most noticeable with 16 processors. Further analysis shows that the bulk of the double or larger invalidations are due to the monitor lock of the distributed loop. Figures 16 and 17 give the invalidation distribution for the 16-processor trace, broken down into monitor lock traffic and all other traffic. Here we note that shared data contributes very little to the invalidations of 2 or more. There are 0.02% that we do not see in the graph, and which are due to occasional collisions in the various data arrays. Unlike SA-TSP, where there are very few data elements, the number of data elements is very large in MP3D and so we do not see any large invalidations. The monitor lock traffic distribution, however, is seen to have significant portions beyond single invalidations. The ratio of time spent doing useful work to time spent in the monitor was found to have an average value of about 16. If there are fewer than about 16 processors, they manage to stagger themselves in the first round of contention. Contention in subsequent rounds is very limited because staggering has occurred. This means that with any more than about 16 processors, we will see a step-increase in invalidations for each processor added. In this manner, a well-behaved program can suddenly produce a very large number of invalidations as it is being scaled.

It is interesting to note that a much faster implementation of the distributed index is possible with some hardware support. This would shift the ratio of unlocked to locked time to a much higher value and would enable the program to be scaled beyond 16 processors. A similar result could be achieved by increasing the grain size — for example by letting each processor move 5 molecules instead of one at a time.

The monitor lock illustrates another phenomenon. When contention for a critical section is low, the lock references cause few invalidations. As more processors are added, the critical section becomes a bottleneck and contention for the lock increases. This in turn raises the number of invalidations caused by lock references. By fixing the program to remove the bottleneck we can also fix the problem of generating a large number of invalidations. In conclusion, synchronization objects themselves are not a problem unless contention for them is high. Since distributed loops and barriers are usually built out of spin locks, this conclusion applies to these synchronization

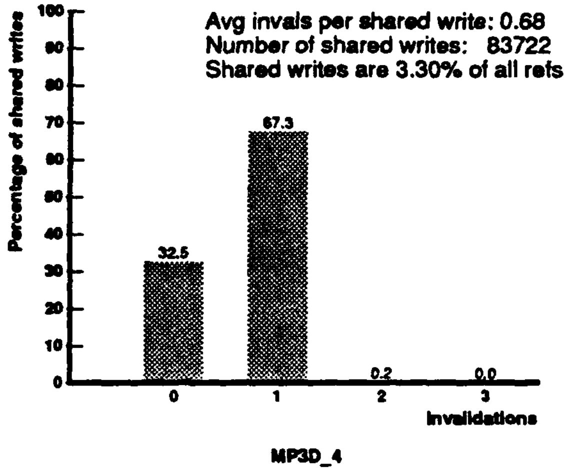


Figure 13: MP3D 4

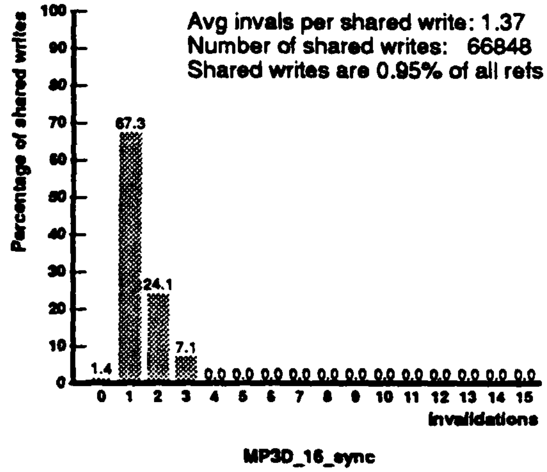


Figure 16: MP3D 16 (Synchronization)

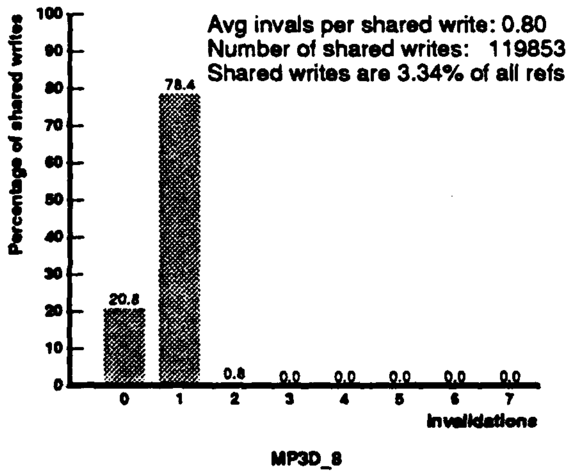


Figure 14: MP3D 8

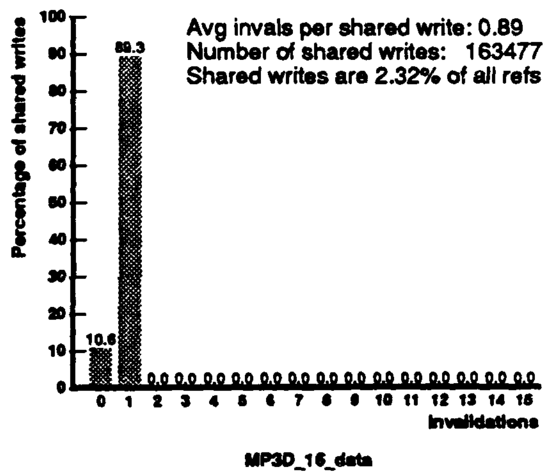


Figure 17: MP3D 16 (Data)

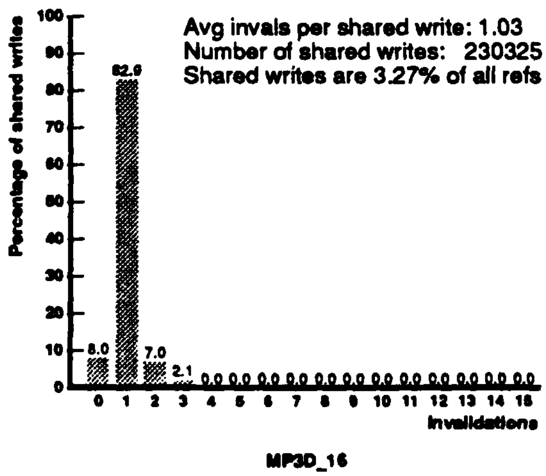


Figure 15: MP3D 16

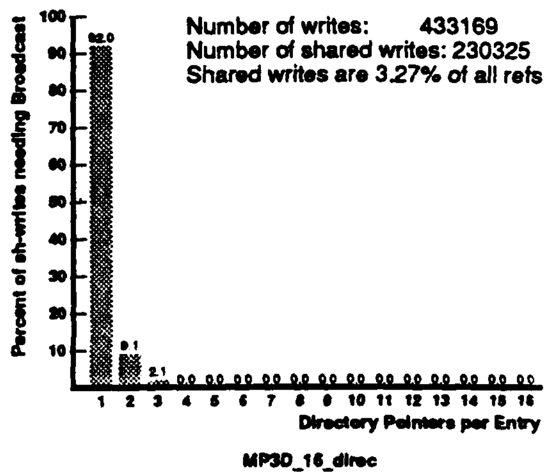


Figure 18: MP3D 16 Directory Performance

objects as well.

Most accesses to shared data by MP3D consist of a read followed immediately by a write. This will allow at most one other cache to be invalidated, unless two processors are accessing the exact same portion of data at the same time. Chances of such a collision are very low and their effect can be tolerated in MP3D, hence no locks are required for the shared data. Update-type data objects such as the shared data of MP3D, can be considered to be a special case of migratory objects, and their invalidation behavior is very similar. The only difference is that each data object is kept for only a short period of time before it moves on to the next processor.

As Figure 18 indicates, directories with just two or three pointers per entry would do extremely well with MP3D. For 3-pointer directory schemes we reduce broadcasts to 2.1% of shared writes, even in the 16-processor case. A re-coding of the distributed loop as suggested above could hold the broadcast percentage to below 1%, even if the number of processors is scaled to well above 16. For MP3D a broadcast fraction of 1% of shared writes corresponds to 0.33 broadcasts per thousand references, which is low enough to be supportable in fairly large machines.

## 6.4 Distributed CSIM

Figures 19, 20 and 21 give the invalidation distributions of Distributed CSIM. We note that the number of shared writes is a much smaller fraction of all references than in the previous three applications. Furthermore, very few shared writes cause more than 2 invalidations. Note that this trace covers a section of code that does not have any synchronization at all, and this is why we do not show a further breakdown of the 16-processor distribution. The distributions we see are for shared data only. Most shared writes cause only zero or single invalidations.

The basic data objects of Distributed CSIM are the element and net structures. Some parts of these structures behave like mostly-read data (e.g., the activation flags) and some parts like migratory data (e.g., next input event pointers). The invalidation patterns vary accordingly.

The activation flag of an element is set as a processor changes one of the element's input values. Many processors can check this flag to see if an element is activated. Later, the element is evaluated and the activation flags are reset. While the *setting* of the activation flag causes only one invalidation, the *resetting* can cause many because many processors may have read the flag in the meantime. The resetting of the activation flags causes about 60% of the shared writes that result in more than single invalidations.

The next input event pointers, on the other hand, are used when an element is being evaluated, and are thus only read and written by one processor while it is updating the element. Hence we see mostly single invalidations - the pattern typical for migratory data.

Another factor that affects the number of invalidations is the connectivity of the circuit being evaluated. Nets that are connected to many elements, clock lines for example, are more likely to cause large invalidations when they are updated.

Figure 22 shows that Distributed CSIM is well suited for directory-based cache schemes. A single-pointer directory captures 17% of broadcasts and a second pointer diminishes this fraction to 3.2%. Further reduction of broadcasts could only be achieved if the program exploited processor locality in some way.

A scaling in the number of processors would result in a larger invalidation average per shared write, but not in more shared writes, since no synchronization objects are present in this portion of Distributed CSIM.

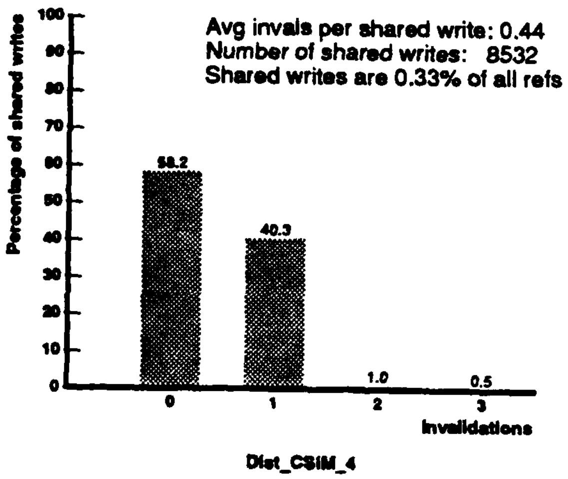


Figure 19: Distributed CSIM 4

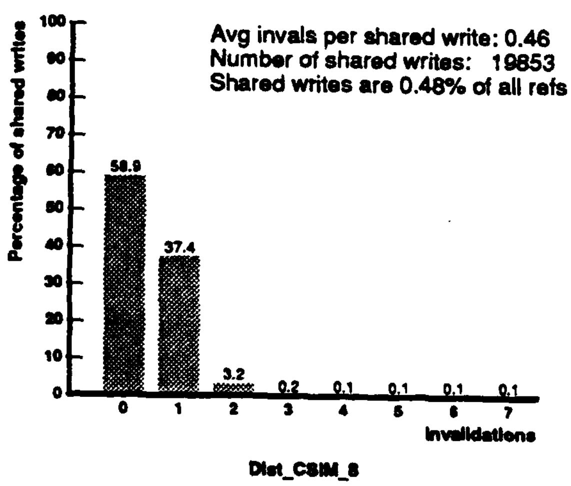


Figure 20: Distributed CSIM 8

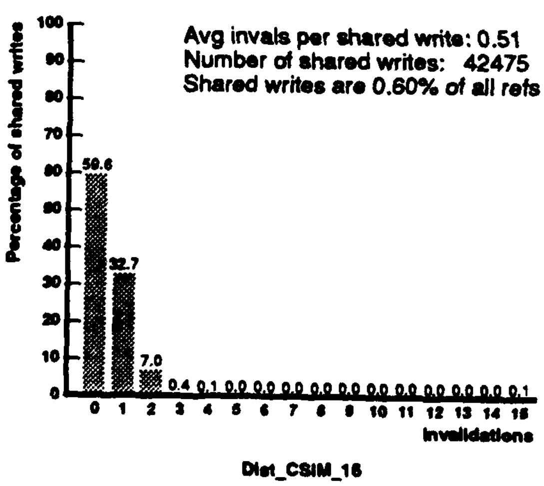


Figure 21: Distributed CSIM 16

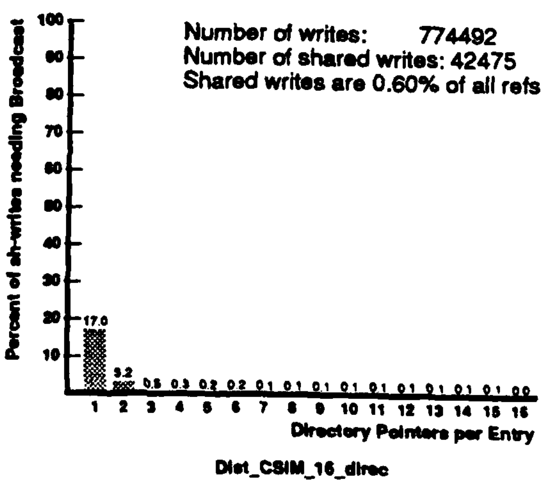


Figure 22: Distributed CSIM 16 Directory Performance

## 6.5 LocusRoute

Figures 23, 24 and 25 show the invalidation distributions for LocusRoute. It is noted that there are very few shared writes per reference. This shows how a well-designed parallel program can avoid excessive interprocess communication. Most of the invalidations are due to data objects. The only synchronization object that shows up is a lock used to control the access to the shared memory allocation routine (ShMalloc).

The single largest source of invalidations is the global cost array. It is a good example of mostly-read data. It is frequently read while testing different routes for a wire, but is written only when the wire route is decided. The average number of invalidations per shared write of the cost array is about 2 with 16 processors, but some writes can cause up to 6 invalidations, depending on how many processors have cached a given portion of the cost array (see Figure 27). Note that there are only 7400 shared writes to the cost array in the 7.7 million reference 16-processor trace.

Invalidations due to the ShMalloc lock are very infrequent in this portion of the program as the program keeps its own free lists and will have allocated most of its shared memory requirement by the time the trace was gathered. As contention for the lock is non-existant, all shared writes to the lock cause only zero or single invalidations (see Figure 26).

LocusRoute would be expected to scale well beyond 16 processors. The shared data is mostly-read and shared writes are very infrequent. As more processors are added, the average number of invalidations per shared write will increase slightly (because more processors are likely to have cached a given portion of the cost array), but the number of shared writes is not expected to increase.

# 7 Generalizations and Conclusions

We have proposed several classes of data objects that can be distinguished by their use in parallel programs and by their invalidation traffic patterns. By merging the invalidation behavior found in the applications discussed in the previous section, we can gain more general insights into the invalidation patterns of certain high-level constructs. We also have the opportunity to predict behavior beyond the 16 processor limit of the case studies.

Little needs to be said about code and read-only data. Since they are never written, they never cause invalidations. Some directory schemes do not allow a memory location to be present in more caches than there are entries (for example $Dir_i NB$ schemes in [2]). This kind of scheme is not suitable for shared code and read-only data.

Migratory data objects move from processor to processor as execution progresses, but they are never manipulated by more than one processor at any one time. The node structures of Maxflow and the global arrays of MP3D are good examples of this data type. Migration of the data object causes at most single invalidations, because each processor writes to the object before relinquishing control of it. Single invalidations are expected, even as the number of processors is scaled. We note that a large number of these invalidations could be avoided if the processors were smart enough to flush the data items out of their cache when they are no longer needed. Hardware and operating system support for this feature seems desirable.

Synchronization primitives were found in all applications. In well-designed applications such as Distributed CSIM and LocusRoute, contention for the critical sections protected by the locks was minimized and this effectively reduced the invalidation traffic caused by the locks. It is seen

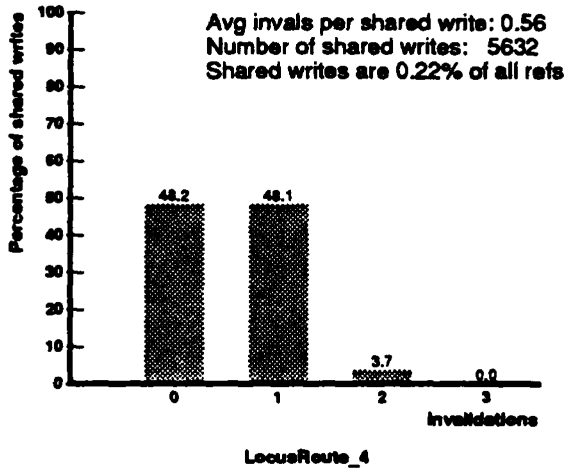


Figure 23: LocusRoute 4

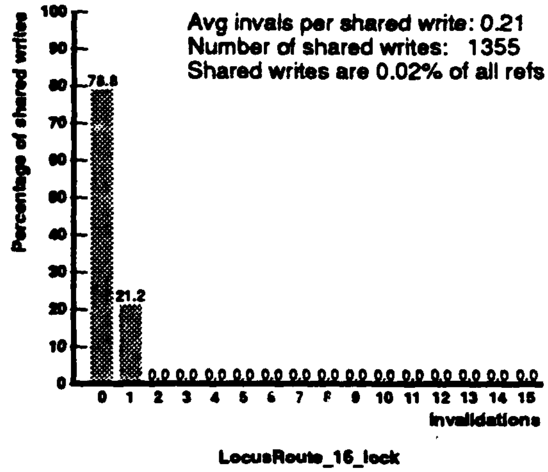


Figure 26: LocusRoute 16 (ShMalloc lock)

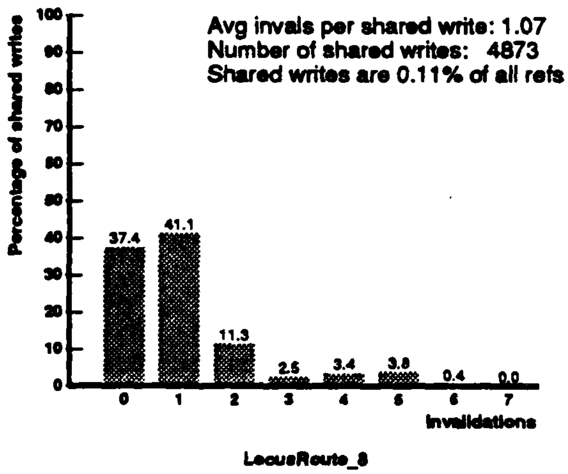


Figure 24: LocusRoute 8

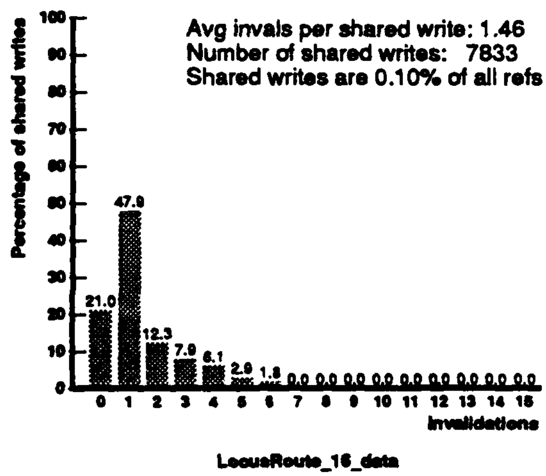


Figure 27: LocusRoute 16 (Data)

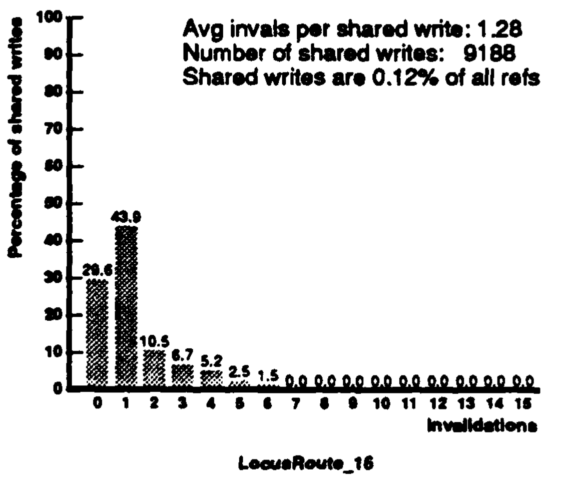


Figure 25: LocusRoute 16

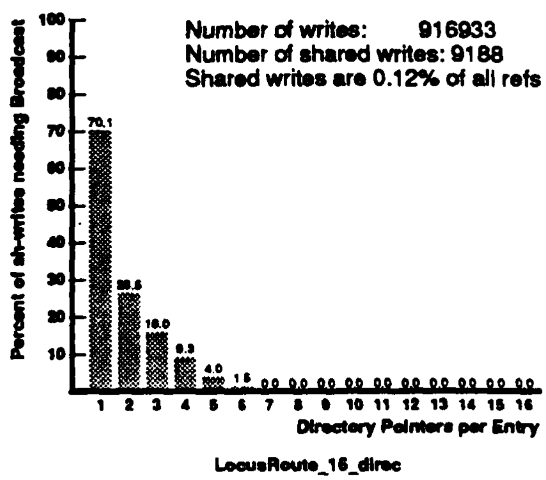


Figure 28: LocusRoute 16 Directory Performance

then. that proper program design will allow the use of locks without a large volume of invalidation traffic. As more processors are used. an ever increasing amount of effort will have to go into the program design to avoid contention over locks. Alternatively. a separate mechanism for dealing with synchronization traffic may be provided.

Mostly-read data such as the global cost array in LocusRoute has potential for causing a large number of invalidations. since each write is preceded by a number of reads from various processors. The average number of invalidations caused by each write is thus high. The good news is that writes to this kind of data tend to be relatively infrequent and hence the total invalidation traffic is not very large. With more processors. we expect an increase in the average number of invalidations per shared write. because it is likely that more processors will have touched the data object before a write to it takes place. Some of this effect may be mitigated by taking advantage of locality. i.e.. assigning work in a local area of the problem to a relatively small section of the processors available. We are currently exploring such issues of locality. which we think will be critical in the design of highly scalable machines.

Frequently read/written data presents the largest problem in terms of invalidations. Not only does each write cause several invalidations. but writes are also frequent. A good example of this type of data is the variable in Maxflow that keeps track of how many processors are waiting on the global queue. Frequently read/written data will show increased invalidations as more processors are used. because more reads and more writes to the data item will take place. This type of data object should be avoided for parallel applications with large number of processors.

In summary. in this paper we have presented data about the invalidation patterns of five applications using 4. 8 and 16 processor traces. By classifying data objects, we are able to predict invalidation behavior beyond the number of processors currently traced. Such extrapolation suggests that directory-based cache schemes with just two or three pointers per entry can work in scalable multiprocessors. if the applications are well-designed. In particular. effort has to be put into limiting contention over synchronization objects and eliminating frequently read/written data objects.

## 8 Acknowledgments

We would like to thank Roberto Bisiani for letting us use his VAX-8350 at CMU and David Black. Robert Baron. and Mary Thompson for helping us with the inner details of the MACH operating system. We wish to thank Larry Soule, Jeff McDonald. Jonathan Rose. Mike Smith and Francisco Carrasco for letting us trace their applications, and for patiently explaining the details of the data structures used by them. We would like to thank Richard Sites of Digital Equipment Corporation. Hudson MA. for providing the VAX-8350 used for tracing at Stanford and for supporting Wolf-Dietrich Weber. Anoop Gupta is supported by DARPA contract N00014-87-K-0828 and by a faculty award from Digital Equipment Corporation.

## References

[1] Anant Agarwal and Anoop Gupta.
Memory Reference Characteristics of Multiprocessor Applications under MACH.
In *ACM SIGMETRICS*. 1988.

[2] Anant Agarwal. Richard Simoni, John Hennessy. and Mark Horowitz.
An Evaluation of Directory Schemes for Cache Coherence.
In *15th International Symposium on Computer Architecture*. 1988.

[3] Francisco Javier Carrasco.
A Parallel Maxflow Implementation.
March 1988.
CS411 - Final Project Report. Stanford University.

[4] M. Censier and P. Feautier.
A New Solution to Coherence Problems in Multicache Systems.
*IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[5] K. M. Chandy and J. Misra.
Asynchronous Distributed Simulation via a Sequence of Parallel Computations.
In *Communications of the ACM*, April 1981.

[6] David Cheriton.
Workform Processing: A Model and Language for Parallel Computation.
Stanford University. Computer Science Technical Report, 1986.

[7] Stephen R. Goldschmidt.
Simulating Multiprocessor Memory Traces.
December 1987.
EE390 Report, Stanford University.

[8] J.R. Goodman.
Using Cache Memory to Reduce Processor-Memory Traffic.
In *Proc. Tenth International Symposium on Computer Architecture*, pages 124–131, June 1983.

[9] Anoop Gupta, Charles Forgy, and Robert Wedig.
Parallel Algorithms and Architectures for Rule-Based Sytems.
In *Proc. 13th Int. Symp. of Computer Architecture*, June 1986.

[10] S. Kirkpatrick, C.D. Gelatt, and M. P. Vecchi.
Optimization by Simulated Annealing.
*Science*. 220(4580):671–680, May 1983.

[11] Lusk, Overbeek, et al.
*Portable Programs for Parallel Processors.*
Holt, Rinehart, and Winston Inc.. 1987.

[12] Lusk, Stevens, and Overbeek.
*A Tutorial on the Use of Monitors in C: Writing Portable Code for Multiprocessors.*
Argonne National Laboratory. Argonne, Illinois 60439. 1986.

[13] Jeffrey D. McDonald.
A Direct Particle Simulation Method for Hypersonic Rarified Flow on a Shared Memory Multiprocessor.
March 1988.
CS411 - Final Project Report. Stanford University.

[14] Jeffrey D. McDonald and Donald Baganoff.
Vectorization of a Particle Simulation Method for Hypersonic Rarified Flow.
In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*. June 1988.

[15] Louis Monier and Pradeep Sindhu.
The Architecture of the Dragon.
In *Proc. Thirtieth IEEE Int. Conference*. pages 118–121. IEEE. Februrary 1985.

[16] R. Katz. S. Eggers. D. Wood. C. Perkins, and R. Sheldon.
Implementing a Cache Consistency Protocol.
In *12th International Symposium on Computer Architecture*. 1985.

[17] Jonathan Rose.
LocusRoute: A Parallel Global Router for Standard Cells.
In *Design Automation Conference*, pages 189–195. June 1988.

[18] Larry Rudolph and Zary Segall.
Dynamic Decentralized Cache Consistency Schemes for MIMD Parallel Processors.
In *Proc. 12th Int. Symp. on Computer Architecture*. pages 355–362. ACM SIGARCH. June 1985.
also SIGARCH Newsletter. Volume 13. Issue 3, 1985.

[19] C. Sechen and A. Sangiovanni-Vincentelli.
The Timberwolf Placement and Routing Package.
*IEEE JSSC*. SC-20(2):510–522. April 1985.

[20] Richard L. Sites and Anant Agarwal.
Multiprocessor Cache Analysis using ATUM.
In *Proc. 15th Annual International Symposium on Computer Architecture*, May 1988.

[21] Michael Smith and Wolf-Dietrich Weber.
Parallel Simulated Annealing.
March 1988.
CS411 - Final Project Report, Stanford University.

[22] Larry Soule and Tom Blank.
Parallel Logic Simulation on General Purpose Machines.
In *Design Automation Conference*, pages 166–171, June 1988.

[23] C. Thacker and L. Stewart.
Firefly: A Multiprocessor Workstation.
In *2nd Int. Conference on Architectural Support for Programming Languages and Operating Systems*. pages 164–172. ACM. October 1987.